Bartok: Support for Systems Programming using Safe Languages

MSR Cool Talk Series
September 28, 2006

David Tarditi
Principal Researcher
Advanced Compiler Technology group
Microsoft Research

History of commercial programming languages (abridged version)

- "C" design parameters (circa 1970)
  - scarce resources
    - PDP-11/20: 16 Kbytes of memory
    - Speed of about 1 MIPS
    - Cost >$10,000 ($50,000 adjusted for inflation)
  - benign environment
  - knowledgeable and trained users
  - barely any connectivity
- Software mistakes meant crashed programs, lost data

The world has changed

- Design parameters (2006)
  - Ample resources (512 Mbytes vs. 8 Kbytes)
  - Hostile environment
  - Untrained users
  - Everything networked
- Software mistakes can mean catastrophic damage
  - Loss of control over your computer
  - Loss or theft of your data
  - Loss of privacy
  - Continual (hidden) monitoring of your computer use
- Example (today)
  - VML Buffer Overrun vulnerability for IE (MS security advisory 925568)
  - 0-day attack enables complete takeover of user's computer
  - Patches being distributed now

What language is used for systems and application programming?

C (circa 1970), with many opportunities for programmer errors

- Buffer overruns
- Manual memory management
- Memory corruption
- No type safety guarantees

C++ (circa 1985), with the same flaws.

Systems and application programs

- Systems programs
  - Operating systems, including their components
    - Device drivers
    - Network stack
    - Middleware (VMs, runtimes, IIS)
    - File systems
    - Browsers
  - Databases
- Applications (end-user oriented)
  - Large complex applications
    - Example: Office
  - Complex enough that they are "systems" themselves
    - With some of the same perf requirements

Cost of lack of safety and affected products

- CERT (since June 1, 2006):
  - At least 23 buffer overrun, memory corruption problems for current patched versions of Microsoft products
  - Products include Windows XP SP2, Windows Server, Word, Excel, Powerpoint, IE 6, Exchange
  - Zero-day exploits circulating
    - Example: VML buffer overrun
    - Visit the wrong Web page or open the wrong document and your computer is taken over
  - CERT issues: #416092 #806548 #650769 #377369 #455516 #884252 #908276 #891204 #6836112 #794580 #394444 #159484 #119180 #159220 #631516 #189140 #936945 #409316 #802324 #303452 #190089 #446012 #608020 #390044 #340060
- Constant patching, updates (plus infrastructure for doing that)
- Unhappy customers

Safe languages

- Safe languages provide strong type safety
  - No buffer overruns, incorrect memory deallocation, memory corruption issues
  - Buffer overruns eliminated using array bounds checking
    - Writing beyond array bounds detected, exception thrown
    - Denial-of-service possible, but not a takeover
  - Memory deallocation issues eliminated using GC
  - Type checking eliminates other sources of memory corruption
- There are plenty of safe languages around
  - C#, Java
  - They are successful!
- May have "unsafe" subset as an escape hatch
  - Example: C#, Modula-3
  - Safety of rest of code depends on correctness of unsafe code.
  - Limits extent of lack of safety

Additional benefits of safe languages

- Eliminates bugs that cause crashes, software failures
- Provides firm foundation for building future software
  - For program analysis
    - Existing tools make significant compromises to deal with C/C++
    - No guarantees that analysis is actually right
  - For showing correctness of programs
  - For specifying program behavior
- Example: Singularity project extended C# to provide checkable message-passing protocols
  - Declare state machine for protocol
  - Check that code actually implements it

Software should be written in safe languages

- Current software written almost entirely in C/C++
  - Despite the <u>proven</u> security problems that causes
- Future software should be written in safe languages
- So why can't we start using safe languages now?!

The issue: today's implementations aren't appropriate

Performance requirements for systems programs include:

- Fast start-up time
  - Consider OS boot
- Excellent code speed
  - Used by millions, worth it
- Low latency
  - For interactive/server apps
- High throughput
- Small memory footprint

Typical implementation of a safe language:

- General-purpose, rich virtual machine with just-in-time compilation

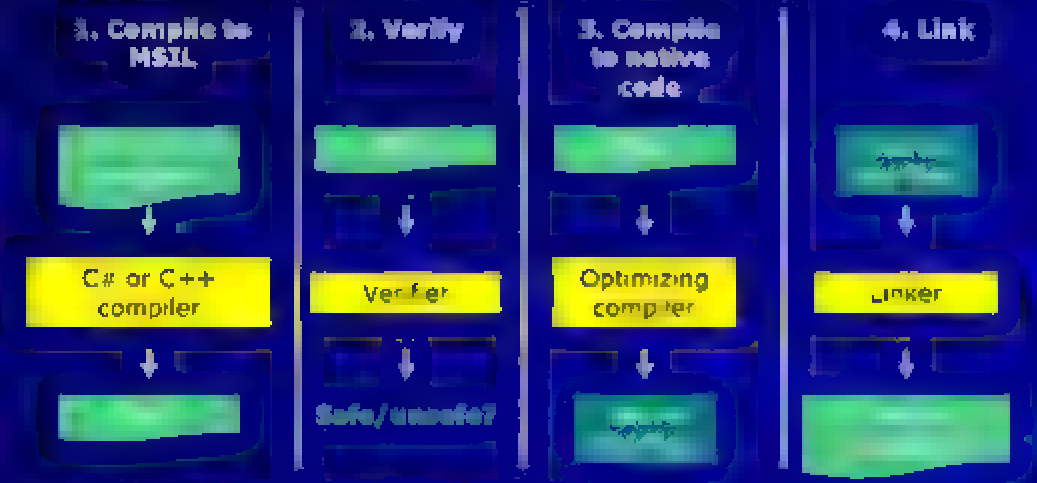
The question

- How do you implement safe languages so that they can be used for systems programming?
- We are investigating this question using Bartok
- Rest of the talk:
  - Design of Bartok
  - Case studies
  - Demo

Bartok design

- MSIL from safe languages as input
  - Eliminate *dynamic* features not needed for systems programming (no JIT, no reflection)
  - Allow unsafe code if necessary.
- Compile to native code
  - Ahead-of-time highly optimizing compiler
  - No runtime overhead, unlike JIT compiler
- Use a lightweight runtime system
  - Minimal, support only key features
  - Customizable
    - Choose GC to match per-app behavior
    - Choose appropriate libraries to run on top of it

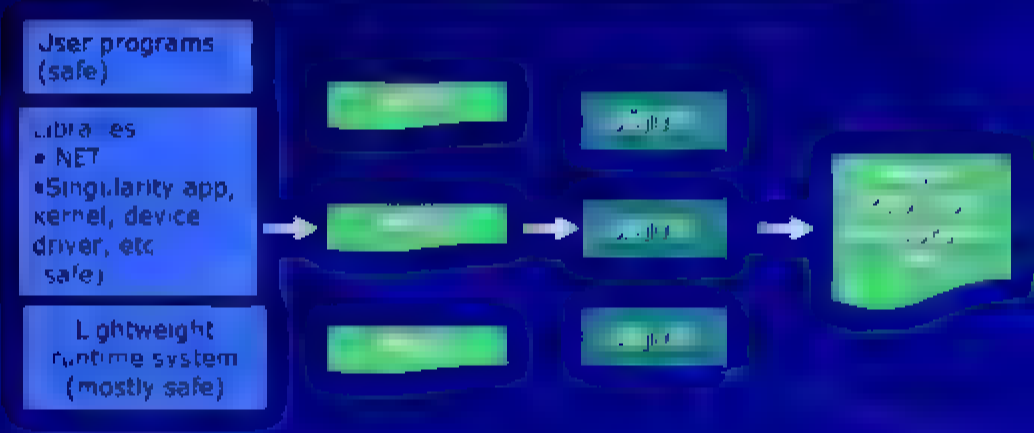
Compiling safe code to native code



Safe = type-safe. No buffer overruns

No memory corruption.

Write as much as possible in safe code — even the runtime



.NET features supported

Supported:

- Core types + object model instructions
  - integers, floats, objects, managed pointers, unmanaged pointers
- Garbage collection, including finalizers, week references
- Exception handling
- Lazy type initialization
- Delegates
- Platform invoke
- Unsafe code
- Data layout attributes
- Generics (coming soon)

<u>Not</u> Supported:

- Dynamic class loading
- Reflection: very limited support
- Code access security
- COM interop

Varying levels of reflection, CAS, COM interop support possible

New features added

- Control over class initialization
  - Require a class constructor be run at process start up
  - Specify initialization order
  - Forbid a class constructor (for sensitive start-up code)
- [NoAlloc] attribute
  - Prohibits allocation in a method and all methods it calls.
  - Enforced by compiler
- [Inline] attribute
  - Force inlining of a function
- Interfaces for assemblies (seperate from implementation)
  - Compile against interface, not actual assembly
  - Avoid undeclared dependencies, break cycles in compilation
- Overlays for type-safe structural casts of arrays of scalars
- Arithmetic on pointer-sized integers, casts between objects and integers (needed for writing GC)

Native interop

- Same support as provided by .NET
  - Pinned objects, P/Invoke for DLL loading, marshalling
- Plus
  - Static linking
  - Use same data layout for native, managed code (no marshalling cost)
    - Auto-generate C++, asm header files
  - Caveat: if native code retains managed pointer, programmer must register it (just like .NET)

Modular runtime system

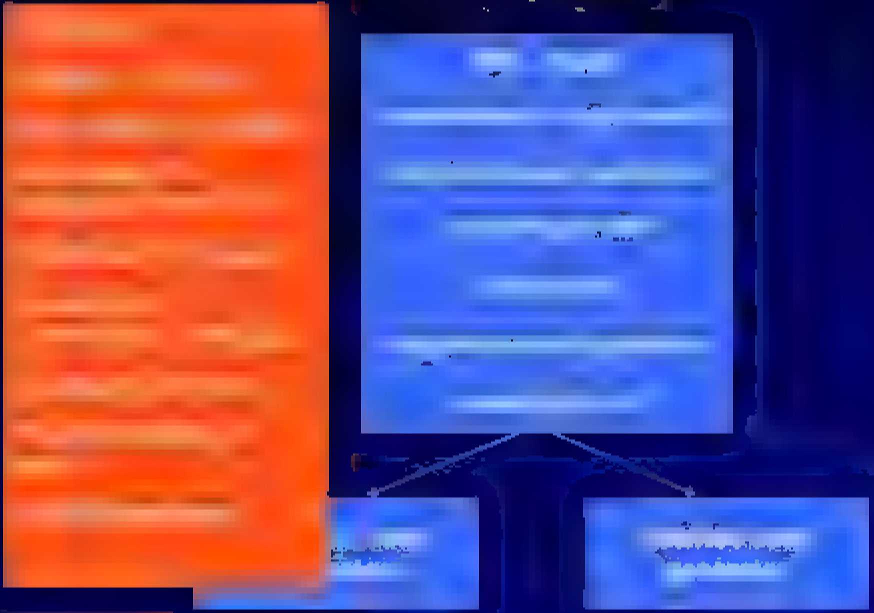
Modular runtime system

- Choose GC, other features to include in a specific runtime
- Allows you to customize the runtime to your application.
- Written entirely in C#
  - See how far we can push *safe* code
  - GC uses unsafe C#
  - Research topic: how to prove safety of unsafe code

Libraries

- Choose the libraries to use with your program.
- It isn't "one size fits all" libraries.
- Bartok supports
  - .NET framework libraries
    - On an "as-needed" basis, we replaced native methods in mscorlib with managed versions.
    - V1.0 and v1.1 support
  - Library for an OS kernel (such as Singularity)
  - Library for Singularity applications

Compiler



Compiler optimizations

- Key to delivering the performance for systems programming
- Choose a different design point than VMs
  - Ahead-of-time compilation, instead of just-in-time compilation
- Benefits:
  - More time for compilation
  - Easier to implement than a JIT
    - No trade-off between compile time and program running time
- Optimizations
  - Includes classic compiler optimizations (about 13)
  - Per-procedure OO/safe language optimizations (about 17)
  - Interprocedural optimizations that are too expensive for just-in-time compilation (about 8)

Classic optimizations

- Scalar optimizations
  - Copy propagation
  - Constant propagation
  - Constant-folding
  - Algebraic simplifications
  - Common-subexpression elimination
  - Loop-invariant removal
  - Dead-code elimination
  - Reverse copy propagation of temporaries
  - Strength reduction of induction variables
- Inlining
- Control-flow optimizations:
  - Jump elimination
  - Short-circuiting
  - Loop header cloning

Object-oriented or managed code optimizations

Also implemented by CLR

- Inlining fast allocation path
- Array-bounds check elimination
- Array store check elimination
- Null check elimination
- Type-test and type cast elimination
- Optimize class initialization
  - Eliminate redundant checks
  - Fast/slow path split
- Redundant field load/store elimination
- Inlining fast allocation path

Additional optimizations

Bartok adds:

- Optimized type test implementation
- Optimize convert operations
  - Widening/narrowing introduced by use of argument stack
- Break interface calls into separate lookup and call operations [CLR does polymorphic inline cache]
- Passing small structs in registers
- Compress garbage collection tables
- Compress exception tables
- Pre-allocate initialized arrays created in class constructors
- Make Thread.GetCurrentThread an intrinsic
- Code instead of large tables for internationalization

Interprocedural optimizations

- These are optimizations across methods
  - Can range from a class to a whole program
  - Larger = more compile time, but better code
  - It's a knob that you dial
- Affects patching/versioning story
  - Perf win ranges from 0-17%
  - May be worth it for some apps and scenarios
    - Examples: kernel

Interprocedural optimizations

- Tree-shaking
  - Eliminate unreachable or unused classes, interfaces, methods, and fields
- Devirtualization of virtual calls
- Eliminate unused formal parameters
- Interprocedural array-bounds check elimination
- Interprocedural array store check elimination
- Interprocedural null check elimination
- Interprocedural type-test and type cast elimination
- Redundant field load/store elimination

Compiler and runtime interaction

Writing runtime in managed code has advantages over using C/C++:

- Dogfooding
  - Forces us to make the compiler better
  - Confront problems of writing systems code ourselves
- Compiler and runtime system have well-defined boundary
  - Compiler reads primitive operations from runtime, inlines them to expose information
    - Examples: allocation sequence, write barriers, type tests, lock operations
  - Simplifies changing runtime
    - Usually knowledge built into compiler
    - Consider supporting 7 GCs that way
  - Few native/managed transitions
- Compiler can automatically specialize the runtime

Tools

- Reuse existing native tools
  - Too costly to build everything on our own.
- Binary looks like C++ executable
- Can use:
  - Existing linker
  - Existing Visual Studio C++ debugger
  - Windbg kernel debugger
  - Existing native profilers

Outline

- Design
- **Case studies**
- Demo

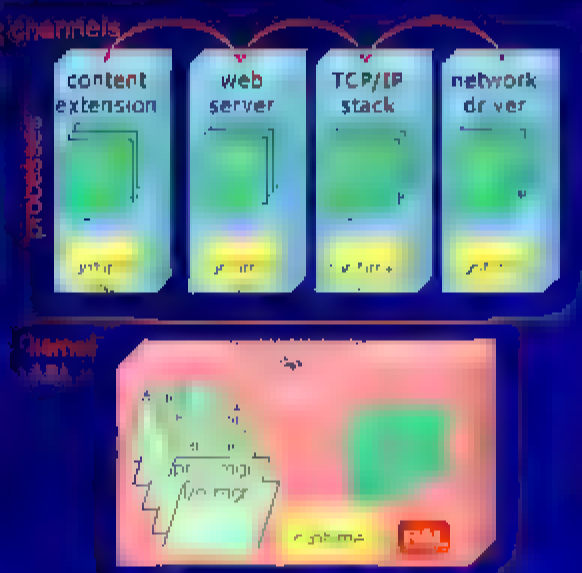
Case study: Singularity Research OS

- *Microsoft Research has written an entire operating system in (mostly) safe code.*
- Goal: develop techniques for building more dependable systems.
- Key approach: use safe languages at all levels
  - Device drivers
  - OS components
  - Applications
- See http://singularity for details

Case study: Singularity Research OS

- Bartok is the compiler and runtime system for Singularity
- We have a working OS
  - Complete with kernel, device drivers, network stack, etc.
  - Boots on real hardware
  - Able to run SpecWeb99

Singularity OS architecture



- Closed Kernel
  - 90% written in C#
  - 17% of files contain unsafe C#
    - 5% of files contain x86 asm or C++
  - OS services in processes
  - device drivers in processes
    - interrupt controller, timer, and debug HAL
  - kernel closed at boot time
- Software isolated processes (SIPs)
  - all user code is verified safe
  - some unsafe code in trusted runtime
  - processes closed at start time
- Safe and efficient communication via strong interfaces
  - channels between processes
  - channel behavior is specified, checked.
  - checked behavior allows us to make communication efficient
- Type safety key to verification & *protection*

Micro Benchmarks

Athlon64 3000+ (1.8GHz) nForce4 SLI	Cost (CPU Cycles)			
	[illegible]	FreeBSD 5.3	Linux 2.6.11 (Red Hat FC4)	Windows XP (SP2)
Minimum kernel API call	[illegible]	878	437	627
Message request/reply	[illegible]	13,300	5,800	(LPC) 4,660 (NP) 6,340
Process create & start	[illegible]	1,030,000	719,000	5,380,000

- Why?
  - All processes run in ring 0
  - Static verification replaces hardware protection
  - Optimizing ahead-of-time compiler (not JIT)

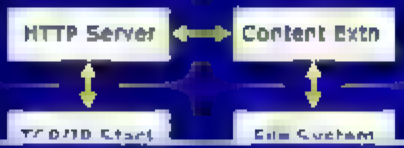
Memory footprint

	Memory footprint "Hello World" process		
	FreeBSD 6.3	Linux 2.6.11 (Red Hat FC4)	Windows XP (SP2)
C - static lib	232K	664K	544K
C++ - static lib	704K	1,216K	572K
C# - w/ GC			3,750K

- C# process w/ GC has similar memory footprint to C++
- minimal process (no GC or exceptions) is ~16K

Macro Benchmark

- SPECweb99 dynamic and static web content benchmark
  - Singularity implementation uses 6 processes:
    - Cassini HTTP Server
    - SPECweb content extension
    - TCP/IP Stack
    - NIC Driver
    - File System
    - Disk Driver



	Response Time ms/op	Throughput op/sec
Singularity	320	247*
Win Server 2003 (IIS)	304	761

*Diskbound. Limited by caching algorithms in experimental FS implementation

Case study: compiling Phoenix

- Phoenix is Microsoft's new compiler and programming tools infrastructure.
- Designed for building static analysis tools and all kinds of compilers (JIT, PreJIT, native C++)

Case study: compiling Phoenix

- Phoenix is written in C++
  - Can compile, link and run as 100% native
    - For example, JIT compiler must be native
  - Or can compile, link and run as 100% managed
    - Managed configurations support plug-in model for user extensions
- Done with homebrew "Dual Mode" technology
  - Implemented with C++-like Object Definition language (PDL) + heavy use of macros

Compiling Phoenix with Bartok

- Took Phoenix compiler and tools infrastructure
- Took c2 client (C++ compiler backend)
- Approximately <u>1.5 million lines of code</u> (including synthesized code)
- Used Bartok to compile <u>managed</u> c2 client, Phoenix DLLs to native code

Results

- Working c2 compiler
  - Passes 1592 regr tests
  - Compiles "phx-all" benchmark (early version of Phoenix sources placed in one file)
- Performance of Phoenix c2 compiling "phx-all"

Version	Exec time	Mem footprint
Native C++	100%	100%
Bartok	137%	158%
CLR (JIT)	197%	126%

•Bartok results can be improved using product quality code generator, product quality GC, additional opts

•CLR results not verified by CLR team

Safety checks are inexpensive

Reductions in execution time from disabling <u>all</u> safety checks

- Singularity SpecWeb99 benchmark: ~5%
- Phoenix c2: ~4%
- 12 IMPACT programs (translated from C++ to C#)
  - Median reduction is ~3%
  - Some are faster than C++ versions

Outline

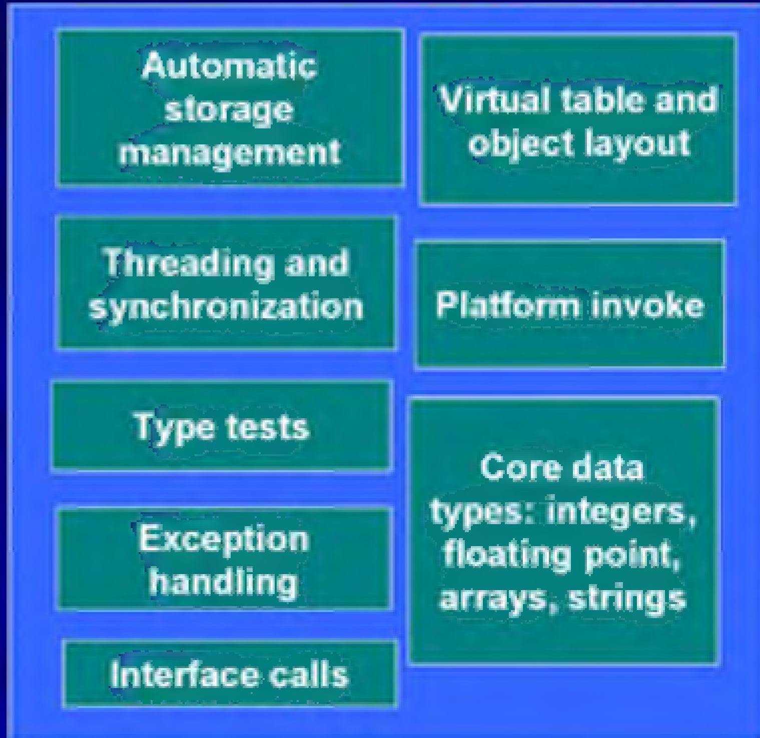
- Design
- Case studies
- Demo

Conclusion and next steps

- We've demonstrated viability of using safe languages for systems programming
  - Singularity
  - Building a large app
- Will help with security, reliability problems for the company and industry.
- Next steps:
  - We're talking to DevDiv Product Units about productization.
  - Their question: are there any customers for this?
  - Would you or your customers want to use a safe language for systems or application programming? Let us know.

History of commercial programming languages (abridged version)

Modular runtime system

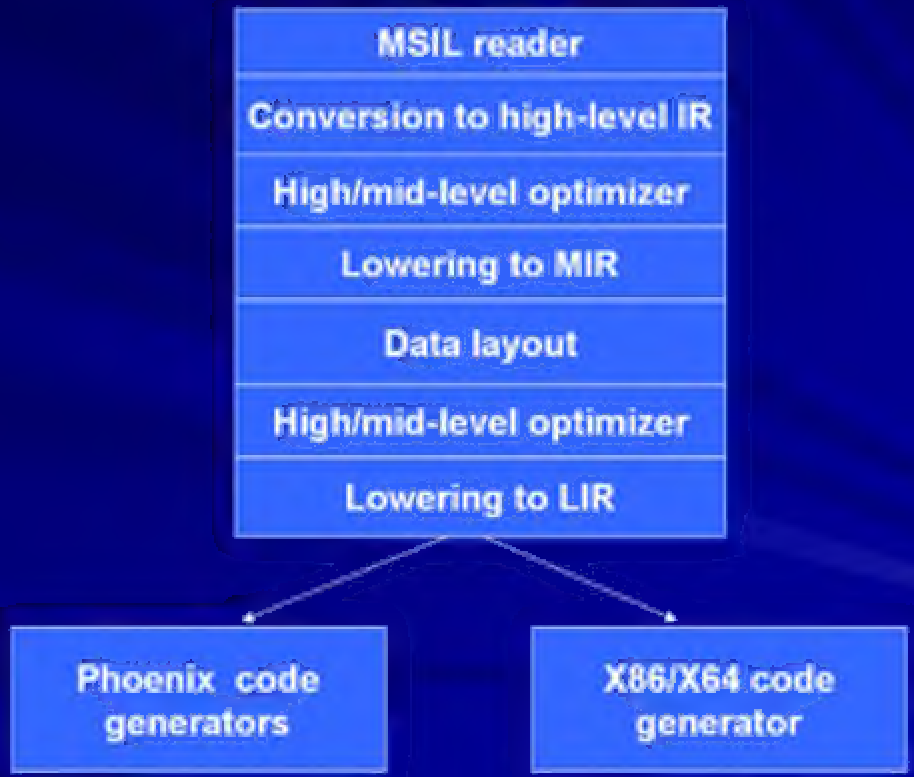


Modular runtime system

1. Gen. semispace
2. Gen. mark-sweep-compact
3. Adaptive version of 1+2
4. Mark-sweep
5. Concurrent mark-sweep
6. Ref counting
7. Deferred ref counting

- Automatic storage management
- Virtual table and object layout
- Threading and synchronization
- Platform invoke
- Type tests
- Core data types: integers, floating point, arrays, strings
- Exception handling
- Interface calls

Compiler


MSIL reader
Conversion to high-level IR
High/mid-level optimizer
Lowering to MIR
Data layout
High/mid-level optimizer
Lowering to LIR
Phoenix code generators
X86/X64 code generator

Macro Benchmark

- SPECweb99 dynamic and static web content benchmark
  - Singularity implementation uses 6 processes:
    - Cassini HTTP Server
    - SPECweb content extension
    - TCP/IP Stack
    - NIC Driver
    - File System
    - Disk Driver